

Applying Slicing Technique to Software Architectures

Jianjun Zhao

Department of Computer Science and Engineering

Fukuoka Institute of Technology

3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0214, Japan

Email: zhao@cs.fit.ac.jp

Abstract

Software architecture is receiving increasingly attention as a critical design level for software systems. As software architecture design resources (in the form of architectural specifications) are going to be accumulated, the development of techniques and tools to support architectural understanding, testing, reengineering, maintenance, and reuse will become an important issue. This paper introduces a new form of slicing, named architectural slicing, to aid architectural understanding and reuse. In contrast to traditional slicing, architectural slicing is designed to operate on the architectural specification of a software system, rather than the source code of a program. Architectural slicing provides knowledge about the high-level structure of a software system, rather than the low-level implementation details of a program. In order to compute an architectural slice, we present the architecture information flow graph which can be used to represent information flows in a software architecture. Based on the graph, we give a two-phase algorithm to compute an architectural slice.

1 Introduction

Software architecture is receiving increasingly attention as a critical design level for software systems [18]. The software architecture of a system defines its high-level structure, exposing its gross organization as a collection of interacting components. A well-defined architecture allows an engineer to reason about system properties at a high level of abstraction. Architectural description languages (ADLs) are formal languages that can be used to represent the architecture of a software system. They focus on the high-level structure of the overall application rather than the implementation details of any specific source module. Recently, a number of architectural description languages have been proposed such as WRIGHT[2], Rapide [13], UniCon [17], and ACME [9] to support formal representation and reasoning of software architectures. As software architecture design resources (in the form of architectural specifications) are going to be accumulated, the development of techniques to support software architectural understanding, testing, reengineering, maintenance and reuse will become an important issue.

One way to support software architecture develop-

ment is to use slicing technique. Program slicing, originally introduced by Weiser [23], is a decomposition technique which extracts program elements related to a particular computation. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. The task to compute program slices is called *program slicing*. To understand the basic idea of program slicing, consider a simple example in Figure 1 which shows: (a) a program fragment and (b) its slice with respect to the slice criterion (Total,14). The slice consists of only those statements in the program that might affect the value of variable Total at line 14. The lines represented by small rectangles are statements that have been sliced away. We refer to this kind of slicing as *traditional slicing* to distinguish it from a new form of slicing introduced later.

Traditional slicing has been studied primarily in the context of conventional programming languages [21]. In such languages, slicing is typically performed by using a control flow graph or a dependence graph [5, 12, 7, 16, 24, 25]. Traditional slicing has many applications in software engineering activities including program understanding [6], debugging [1], testing [3], maintenance [8], reuse [15], reverse engineering [4], and complexity measurement [16].

Applying slicing technique to software architectures promises benefit for software architecture development at least in two aspects. First, architectural understanding and maintenance should benefit from slicing. When a maintainer wants to modify a component in a software architecture in order to satisfy new design requirements, the maintainer must first investigate which components will affect the modified component and which components will be affected by the modified component. This process is usually called *impact analysis*. By slicing a software architecture, the maintainer can extract the parts of a software architecture containing those components that might affect, or be affected by, the modified component. The slicing tool which provides such information can assist the maintainer greatly. Second, architectural reuse should benefit from slicing. While reuse of code is important, in order to make truly large gains in productivity and quality, reuse of software designs and patterns may offer the greater potential for return on investment. By slicing a software architecture, a sys-

```

(a) A program fragment.
1  begin
2    read(X,Y);
3    Total := 0.0;
4    Sum := 0.0;
5    if X <= 1 then
6      Sum := Y;
7    else
8      begin
9        read(Z);
10       Total := X * Y;
11     end;
12   end if
13   Write(Total, sum);
14 end

(b) a slice of (a) on the criterion (Total,14)
1  begin
2    read(X,Y);
3    Total := 0.0;
4    [ ]
5    if X <= 1 then
6      [ ]
7    else
8      begin
9        [ ]
10       Total := X * Y;
11     end;
12   end if
13   [ ]
14 end

```

Figure 1: A program fragment and its slice on criterion (Total,14).

tem designer can extract reusable architectures from it, and reuse them into new system designs for which they are appropriate.

While slicing is useful in software architecture development, existing slicing techniques for conventional programming languages can not be applied to architectural specifications straightforwardly due to the following reasons. Generally, the traditional definition of slicing is concerned with slicing programs written in conventional programming languages which primarily consist of variables and statements, and the slicing notions are usually defined as (1) a slicing criterion is a pair (s, V) where s is a statement and V is a set of variables defined or used at s , and (2) a slice consists of only statements. However, in a software architecture, the basic elements are components and their interconnections, but neither variables nor statements as in conventional programming languages. Therefore, to perform slicing at the architectural level, new slicing notions for software architectures must be defined.

In this paper, we introduce a new form of slicing, named *architectural slicing*. In contrast to traditional slicing, architectural slicing is designed to operate on

a formal architectural specification of a software system, rather than the source code of a conventional program. Architectural slicing provides knowledge about the high-level structure of a software system, rather than the low-level implementation details of a conventional program. Our purpose for development of architectural slicing is different from that for development of traditional slicing. While traditional slicing was designed originally for supporting source code level understanding and debugging of conventional programs, architectural slicing was primarily designed for supporting architectural level understanding and reuse of large-scale software systems. However, just as traditional slicing has many other applications in software engineering activities, we believe that architectural slicing is also useful in other software architecture development activities including architectural testing, reverse engineering, reengineering, and complexity measurement.

Abstractly, our slicing algorithm takes as input a formal architectural specification (written in its associated architectural description language) of a software system, then it removes from the specification those components and interconnections between components which are not necessary for ensuring that the semantics of the specification of the software architecture is maintained. This benefit allows unnecessary components and interconnections between components to be removed at the architectural level of the system which may lead to considerable space savings, especially for large-scale software systems whose architectures consist of numerous components. In order to compute an architectural slice, we present the *architecture information flow graph* which can be used to represent information flows in a software architecture. Based on the graph, we give a two-phase algorithm to compute an architectural slice.

The rest of the paper is organized as follows. Section 2 briefly introduces how to represent a software architecture using WRIGHT: an architectural description language. Section 3 shows a motivation example. Section 4 defines some notions about slicing software architectures. Section 5 presents the architecture information flow graph for software architectures. Section 6 gives a two-phase algorithm for computing an architectural slice. Section 7 discusses the related work. Concluding remarks are given in Section 8.

2 Software Architectural Specification in WRIGHT

We assume that readers are familiar with the basic concepts of software architecture and architectural description language, and in this paper, we use WRIGHT architectural description language [2] as our target language for formally representing software architectures. The selection of WRIGHT is based on that it supports to represent not only the architectural structure but also the architectural behavior of a software architecture.

Below, we use a simple WRIGHT architectural specification taken from [14] as a sample to briefly introduce

```

Configuration GasStation
  Component Customer
    Port Pay = pay!x → Pay
    Port Gas = take → pump?x → Gas
    Computation = Pay.pay!x → Gas.take → Gas.pump?x → Computation
  Component Cashier
    Port Customer1 = pay?x → Customer1
    Port Customer2 = pay?x → Customer2
    Port Topump = pump!x → Topump
    Computation = Customer1.pay?x → Topump.pump!x → Computation
    [] Customer2.pay?x → Topump.pump!x → Computation
  Component Pump
    Port Oil1 = take → pump!x → Oil1
    Port Oil2 = take → pump!x → Oil2
    Port Fromcashier = pump?x → Fromcashier
    Computation = Fromcashier.pump?x →
      (Oil1.take → Oil1.pump!x → Computation)
      [] (Oil2.take → Oil2.pump!x → Computation)
  Connector Customer_Cashier
    Role Givemoney = pay!x → Givemoney
    Role Getmoney = pay?x → Getmoney
    Glue = Givemoney.pay?x → Getmoney.pay!x → Glue
  Connector Customer_Pump
    Role Getoil = take → pump?x → Getoil
    Role Giveoil = take → pump!x → Giveoil
    Glue = Getoil.take → Giveoil.take → Giveoil.pump?x → Getoil.pump!x → Glue
  Connector Cashier_Pump
    Role Tell = pump!x → Tell
    Role Know = pump?x → Know
    Glue = Tell.pump?x → Know.pump!x → Glue
  Instances
    Customer1: Customer
    Customer2: Customer
    cashier: Cashier
    pump: Pump
    Customer1_cashier: Customer_Cashier
    Customer2_cashier: Customer_Cashier
    Customer1_pump: Customer_Pump
    Customer2_pump: Customer_Pump
    cashier_pump: Cashier_Pump
  Attachments
    Customer1.Pay as Customer1_cashier.Givemoney
    Customer1.Gas as Customer1_pump.Getoil
    Customer2.Pay as Customer2_cashier.Givemoney
    Customer2.Gas as Customer2_pump.Getoil
    cashier.Customer1 as Customer1_cashier.Getmoney
    cashier.Customer2 as Customer2_cashier.Getmoney
    cashier.Topump as cashier_pump.Tell
    pump.Fromcashier as cashier_pump.Know
    pump.Oil1 as Customer1_pump.Giveoil
    pump.Oil2 as Customer2_pump.Giveoil
End GasStation.

```

Figure 2: An architectural specification in WRIGHT.

how to use WRIGHT to represent a software architecture. The specification is showed in Figure 2 which models the system architecture of a Gas Station system [11].

2.1 Representing Architectural Structure

WRIGHT uses a *configuration* to describe architectural structure as graph of components and connectors.

Components are computation units in the system. In WRIGHT, each component has an *interface* defined by a set of *ports*. Each port identifies a point of interaction between the component and its environment.

Connectors are patterns of interaction between components. In WRIGHT, each connector has an *interface* defined by a set of *roles*. Each role defines a participant of the interaction represented by the connector.

A WRIGHT architectural specification of a system is

defined by a set of component and connector type definitions, a set of instantiations of specific objects of these types, and a set of *attachments*. Attachments specify which components are linked to which connectors.

For example, in Figure 2 there are three component type definitions, *Customer*, *Cashier* and *Pump*, and three connector type definitions, *Customer_Cashier*, *Customer_Pump* and *Cashier_Pump*. The configuration is composed of a set of instances and a set of attachments to specify the architectural structure of the system.

2.2 Representing Architectural Behavior

WRIGHT models architectural behavior according to the significant events that take place in the computation of components, and the interactions between com-

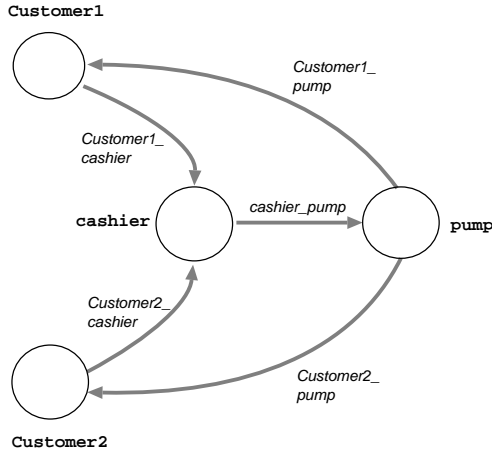


Figure 3: The architecture of the Gas Station system.

ponents as described by the connectors. The notation for specifying event-based behavior is adapted from CSP [10]. Each CSP process defines an alphabet of events and the permitted patterns of events that the process may exhibit. These processes synchronize on common events (i.e., interact) when composed in parallel. WRIGHT uses such process descriptions to describe the behavior of ports, roles, computations and glues.

A *computation* specification specifies a component's behavior: the way in which it accepts certain events on certain *ports* and produces new events on those or other ports. Moreover, WRIGHT uses an overbar to distinguish initiated events from observed events *. For example, the **Customer** initiates Pay action (i.e., pay!x) while the **Cashier** observes it (i.e., pay?x).

A *port* specification specifies the local protocol with which the component interacts with its environment through that port.

A *role* specification specifies the protocol that must be satisfied by any port that is attached to that role. Generally, a port need not have the same behavior as the role that it fills, but may choose to use only a subset of the connector capabilities. For example, the **Customer** role Gas and the **Customer_Pump** port Getoil are identical.

A *glue* specification specifies how the roles of a connector interact with each other. For example, a **Cashier_Pump** tell (Tell.pump?x) must be transmitted to the **Cashier_Pump** know (Know.pump!x).

As a result, based on formal WRIGHT architectural specifications, we can infer which ports of a component are input ports and which are output ports. Also, we can infer which roles are input roles and which are output roles. Moreover, the direction in which the information transfers between ports and/or roles can also

be inferred based on the formal specification. As we will show in Section 5, such kinds of information can be used to construct the information flow graph for a software architecture for computing an architectural slice efficiently.

In order to focus on the key ideas of architectural slicing, in this paper we assume that a software architecture be represented by a formal architectural specification which contains three basic types of design entities, namely, *components* whose interfaces are defined by a set of elements called *ports*, *connectors* whose interfaces are defined by a set of elements called *roles* and the *configuration* whose topology is declared by a set of elements called *instances* and *attachments*. Moreover, each component has a special element called *computation* and each connector has a special element called *glue* as we described above.

In the rest of the paper, we assume that an architectural specification P be denoted by (C_m, C_n, c_g) where:

- C_m is the set of components in P ,
- C_n is the set of connectors in P , and
- c_g is the configuration of P .

3 Motivation Example

We present a simple example to explain our approach on architectural slicing. The example also shows one application of architectural slicing, in which it is used in the impact analysis of software architectures.

Consider the Gas Station system whose architectural representation is shown in Figure 3, and WRIGHT specification is shown in Figure 2. Suppose a maintainer needs to modify the component **cashier** in the architectural specification in order to satisfy some new design requirements. The first thing the maintainer has to do is to investigate which components and connectors interact with component **cashier** through its ports **Customer1**, **Customer2**, and **Topump**. A common way is to manually check the source code of the specification to find such information. However, it is very time-consuming and error-prone even for a small size specification because there may be complex dependence relations between components in the specification. If the maintainer has an architectural slicer at hand, the work may probably be simplified and automated without the disadvantages mentioned above. In such a scenario, an architectural slicer is invoked, which takes as input: (1) a complete architectural specification of the system, and (2) a set of ports of the component **cashier**, i.e., **Customer1**, **Customer2** and **Topump** (this is an *architectural slicing criterion*). The slicer then computes a backward and forward architectural slice respectively with respect to the criterion and outputs them to the maintainer. A backward architectural slice is a partial specification of the original one which includes those

*In this paper, we use an underbar to represent an initiated event instead of an overbar that used in the original WRIGHT language definition [2].

components and connectors that might affect the component **cashier** through the ports in the criterion, and a forward architectural slice is a partial specification of the original one which includes those components and connectors that might be affected by the component **cashier** through the ports in the criterion. The other parts of the specification that might not affect or be affected by the component **cashier** will be removed, i.e., sliced away from the original specification. The maintainer can thus examine only the contents included in a slice to investigate the impact of modification. Using the algorithm we will present in Section 6, the slice shown in Figure 6 can be computed.

4 Architectural Slicing

Intuitively, an *architectural slice* may be viewed as a subset of the behavior of a software architecture, similar to the original notion of the traditional static slice. However, while a traditional slice intends to isolate the behavior of a specified set of program variables, an architectural slice intends to isolate the behavior of a specified set of a component or connector's elements. Given an architectural specification $P = (C_m, C_n, c_g)$, our goal is to compute an architectural slice $S_p = (C'_m, C'_n, c'_g)$ which should be a "sub-architecture" of P and preserve partially the semantics of P . To define the meanings of the word "sub-architecture," we introduce the concepts of a reduced component, connector and configuration.

Definition 4.1 Let $P = (C_m, C_n, c_g)$ be an architectural specification and $c_m \in C_m$, $c_n \in C_n$, and c_g be a component, connector, and configuration of P respectively:

- A reduced component of c_m is a component c'_m that is derived from c_m by removing zero, or more elements from c_m .
- A reduced connector of c_n is a connector c'_n that is derived from c_n by removing zero, or more elements from c_n .
- A reduced configuration of c_g is a configuration c'_g that is derived from c_g by removing zero, or more elements from c_g .

The above definition showed that a reduced component, connector, or configuration of a component, connector, or configuration may equal itself in the case that none of its elements has been removed, or an *empty* component, connector, or configuration in the case that all its elements have been removed.

For example, the followings show a component **Customer**, a connector **Customer_Cashier**, and a configuration as well as their corresponding reduced component, connector, and configuration. The small rectangles represent those ports, roles, or instances and attachments that have been removed from the original

component, connector, or configuration.

- (1) The component **Customer** and its reduced component (with * mark) in which the port **Gas** and elements **Gas.take** and **Gas.pump?x** that are related to **Gas** in the computation have been removed.

Component Customer
Port Pay = pay!x → Pay
Port Gas = take → pump?x → Gas
Computation = Pay.pay!x → Gas.take
→ Gas.pump?x → Computation

* **Component Customer**
Port Pay = pay!x → Pay
□□□□□□□□□□□□□□□□□□
Computation = Pay.pay!x → □□□□□□
→ □□□□□□ → Computation

- (2) The connector **Customer_Cashier** and its reduced connector (with * mark) in which the role **Givemoney** and the element **Givemoney.pay?x** that is related to **Givemoney** in the glue have been removed.

Connector Customer_Cashier
Role Givemoney = pay!x → Givemoney
Role Getmoney = pay?x → Getmoney
Glue = Givemoney.pay?x → Getmoney.pay!x
→ Glue

* **Connector Customer_Cashier**
□□□□□□□□□□□□□□□□□□
Role Getmoney = pay?x → Getmoney
Glue = □□□□□□□□□□ → Getmoney.pay!x
→ Glue

- (3) The configuration and its reduced configuration (with * mark) in which some instances and attachments have been removed.

Instances

Customer1: Customer
Customer2: Customer
cashier: Cashier
pump: Pump
Customer1_cashier: Customer_Cashier
Customer2_cashier: Customer_Cashier
Customer1_pump: Customer_Pump
Customer2_pump: Customer_Pump
cashier_pump: Cashier_Pump

Attachments

Customer1.Pay as Customer1_cashier.Givemoney
Customer1.Gas as Customer1_pump.Getoil
Customer2.Pay as Customer2_cashier.Givemoney
Customer2.Gas as Customer2_pump.Getoil
casier.Customer1 as Customer1_cashier.Getmoney
casier.Customer2 as Customer2_cashier.Getmoney
cashier.Topump as cashier.pump.Tell
pump.Fromcashier as cashier.pump.Know
pump.Oil1 as Customer1_pump.Giveoil
pump.Oil2 as Customer2_pump.Giveoil

*** Instances**

```
Customer1: Customer
Customer2: Customer
cashier: Cashier
□□□□□□□□
Customer1_cashier: Customer_Cashier
Customer2_cashier: Customer_Cashier
□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□
```

*** Attachments**

```
Customer1.Pay as Customer1_cashier.Givemoney
□□□□□□□□□□□□□□□□□□□□□□□□
Customer2.Pay as Customer2_cashier.Givemoney
□□□□□□□□□□□□□□□□□□□□□□□□
casier.Customer1 as Customer1_cashier.Getmoney
casier.Customer2 as Customer2_cashier.Getmoney
□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□
```

Having the definitions of a reduced component, connector and configuration, we can define the meaning of the word “sub-architecture”.

Definition 4.2 Let $P = (C_m, C_n, c_g)$ and $P' = (C'_m, C'_n, c'_g)$ be two architectural specifications. Then P' is a reduced architectural specification of P if:

- $C'_m = \{c'_{m_1}, c'_{m_2}, \dots, c'_{m_k}\}$ is a “subset” of $C_m = \{c_{m_1}, c_{m_2}, \dots, c_{m_k}\}$ such that for $i = 1, 2, \dots, k$, c'_{m_i} is a reduced component of c_{m_i} ,
- $C'_n = \{c'_{n_1}, c'_{n_2}, \dots, c'_{n_k}\}$ is a “subset” of $C_n = \{c_{n_1}, c_{n_2}, \dots, c_{n_k}\}$ such that for $i = 1, 2, \dots, k$, c'_{n_i} is a reduced connector of c_{n_i} ,
- c'_g is a reduced configuration of c_g ,

Having the definition of a reduced architectural specification, we can define some notions about slicing software architectures.

In a WRIGHT architectural specification, for example, a component’s interface is defined to be a set of ports which identify the form of the component interacting with its environment, and a connector’s interface is defined to be a set of roles which identify the form of the connector interacting with its environment. To understand how a component interacts with other components and connectors for making changes, a maintainer must examine each port of the component of interest. Moreover, it has been frequently emphasized that connectors are as important as components for architectural design, and a maintainer may also want to modify a connector during the maintenance. To satisfy these requirements, for example, we can define a slicing criterion for a WRIGHT architectural specification as a set of ports of a component or a set of roles of a connector of interest.

Definition 4.3 Let $P = (C_m, C_n, c_g)$ be an architectural specification. A slicing criterion for P is a pair (c, E) such that:

1. $c \in C_m$ and E is a set of elements of c , or
2. $c \in C_n$ and E is a set of elements of c .

Note that the selection of a slicing criterion depends on users’ interests on what they want to examine. If they are interested in examining a component in an architectural specification, they may use slicing criterion 1. If they are interested in examining a connector, they may use slicing criterion 2. Moreover, the determination of the set E also depends on users’ interests on what they want to examine. If they want to examine a component, then E may be the set of ports or just a subset of ports of the component. If they want to examine a connector, then E may be the set of roles or just a subset of roles of the connector.

Definition 4.4 Let $P = (C_m, C_n, c_g)$ be an architectural specification.

- A backward architectural slice $S_{bp} = (C'_m, C'_n, C'_g)$ of P on a given slicing criterion (c, E) is a reduced architectural specification of P which contains only those reduced components, connectors, and configuration that might directly or indirectly affect the behavior of c through elements in E .
- Backward-slicing an architectural specification P on a given slicing criterion is to find the backward architectural slice of P with respect to the criterion.

Definition 4.5 Let $P = (C_m, C_n, c_g)$ be an architectural specification.

- A forward architectural slice $S_{fp} = (C'_m, C'_n, C'_g)$ of P on a given slicing criterion (c, E) is a reduced architectural specification of P which contains only those reduced components, connectors, and configuration that might be directly or indirectly affected by the behavior of c through elements in E .
- Forward-slicing an architectural specification P on a given slicing criterion is to find the forward architectural slice of P with respect to the criterion.

From Definitions 4.4 and 4.5, it is obviously that there is at least one backward slice and at least one forward slice of an architectural specification that is the specification itself. Moreover, the architecture represented by S_{bp} or S_{fp} should be a “sub-architecture” of the architecture represented by P .

Defining an architectural slice as a reduced architectural specification of the original one is particularly useful for supporting architectural reuse. By using an architectural slicer, a system designer can automatically decompose an existing architecture (in the case that its architectural specification is available) into some small architectures each having its own functionality which may be reused in new system designs. Moreover, the view of an architectural slice as a reduced architectural specification does not reduce its usefulness when applied

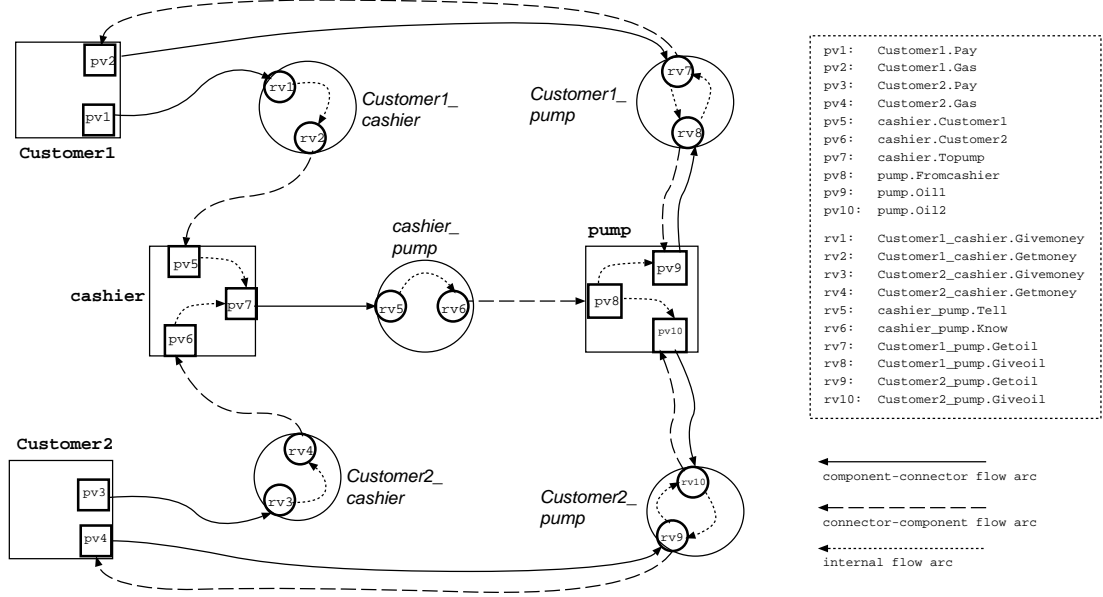


Figure 4: The information flow graph of the architectural specification in Figure 2.

it to architectural understanding because it also contains enough information for a maintainer to facilitate the modification.

5 The Information Flow Graph for Software Architectures

In this section, we present the architecture information flow graph for software architectures on which architectural slices can be computed efficiently.

The architecture information flow graph is an arc-classified digraph whose vertices represent the ports of components and the roles of the connectors in an architectural specification, and arcs represent possible information flows between components and/or connectors in the specification.

Definition 5.1 *The Architecture Information Flow Graph (AIFG) of an architectural specification P is an arc-classified digraph $(V_{com}, V_{con}, Com, Con, Int)$, where:*

- V_{com} is the set of port vertices of P ;
- V_{con} is the set of role vertices of P ;
- Com is the set of component-connector flow arcs;
- Con is the set of connector-component flow arcs;
- Int is the set of internal flow arcs.

There are three types of information flow arcs in the AIFG, namely, *component-connector flow arcs*, *connector-component flow arcs*, and *internal flow arcs*.

Component-connector flow arcs are used to represent information flows between a port of a component and a role of a connector in an architectural specification. Informally, if there is an information flow from a port of a component to a role of a connector in the specification, then there is a component-connector flow arc in the AIFG which connects the corresponding port vertex to the corresponding role vertex. For example, from the WRIGHT specification shown in Figure 2, we can know that there is an information flow from the port Topump of the component cashier to the role Tell of the connector cashier_pump. Therefore there is a component-connector flow arc in the AIFG in Figure 4 which connects the port vertex of port Topump to the role vertex of role Tell.

Connector-component flow arcs are used to represent information flows between a role of a connector and a port of a component in an architectural specification. Informally, if there is an information flow from a role of a connector to a port of a component in the specification, then there is a connector-component flow arc in the AIFG which connects the corresponding role vertex to the corresponding port vertex. For example, from the WRIGHT specification in Figure 2, we can know that there is an information flow from the role Know of the connector cashier_pump to the port Fromcashier of the component pump. Therefore, there is a connector-component flow arc in the AIFG in Figure 4 which connects the role vertex for role Know to the port vertex for port Fromcashier.

Internal flow arcs are used to represent internal information flows within a component or connector in an architectural specification. Informally, for a component in the specification, there is an internal flow from an

input port to an output port, and for a connector in the specification, there is an internal flow from an input role to an output role. For example, in Figure 2, there is an internal flow from the role `Givemoney` to the role `Getmoney` of the connector `Customer1_cashier` and also an internal flow arc from the port `Fromcashier` to the port `Oil1` of component `pump`.

As we introduced in Section 2, `WRIGHT` uses CSP-based model to specify the behavior of a component and a connector of a software architecture. `WRIGHT` allows user to infer which ports of a component are input and which are output, and which roles of a connector are input and which are output based on a `WRIGHT` architectural specification. Moreover, it also allows user to infer the direction in which the information transfers between ports and/or roles. As a result, by using a static analysis tool which takes an architectural specification as its input, we can construct the AIFG of a `WRIGHT` architectural specification automatically.

Figure 4 shows the AIFG of the architectural specification in Figure 2. In the figure, large squares represent components in the specification, and small squares represent the ports of each component. Each port vertex has a name described by `component_name.port_name`. For example, `pv5 (cashier.Customer1)` is a port vertex that represents the port `Customer1` of the component `cashier`. Large circles represent connectors in the specification, and small circles represent the roles of each connector. Each role vertex has a name described by `connector_name.role_name`. For example, `rv5 (cashier_pump.Tell)` is a role vertex that represents the role `Tell` of the connector `cashier_pump`. The complete specification of each vertex is shown on the right side of the figure.

Solid arcs represent component-connector flow arcs that connect a port of a component to a role of a connector. Dashed arcs represent connector-component flow arcs that connect a role of a connector to a port of a component. Dotted arcs represent internal flow arcs that connect two ports within a component (from an input port to an output port), or two roles within a connector (from an input role to an output role). For example, $(rv2, pv5)$ and $(rv6, pv8)$ are connector-component flow arcs. $(pv7, rv5)$ and $(pv9, rv8)$ are component-connector flow arcs. $(rv1, rv2)$ and $(pv8, pv10)$ are internal flow arcs.

6 Computing Architectural Slices

The slicing notions defined in Section 4 give us only a general view of an architectural slice, and do not tell us how to compute it. In this section we present a two-phase algorithm to compute a slice of an architectural specification based on its information flow graph. Our algorithm contains two phases: (1) Computing a slice S_g over the information flow graph of an architectural specification, and (2) Constructing an architectural slice S_p from S_g .

6.1 Computing a Slice over the AIFG

Let $P = (C_m, C_n, c_g)$ be an architectural specification and $G = (V_{com}, V_{con}, Com, Con, Int)$ be the AIFG of P . To compute a slice over the G , we refine the slicing notions defined in Section 4 as follows:

- A slicing criterion for G is a pair (c, V_c) such that: (1) $c \in C_m$ and V_c is a set of port vertices corresponding to the ports of c , or (2) $c \in C_n$ and V_c is a set of role vertices corresponding to roles of c .
- The backward slice $S_{bg}(c, V_c)$ of G on a given slicing criterion (c, V_c) is a subset of vertices of G such that for any vertex v of G , $v \in S_{bg}(c, V_c)$ iff there exists a path from v to $v' \in V_c$ in the AIFG.
- The forward slice $S_{fg}(c, V_c)$ of G on a given slicing criterion (c, V_c) is a subset of vertices of G such that for any vertex v of G , $v \in S_{fg}(c, V_c)$ iff there exists a path from $v' \in V_c$ to v in the AIFG.

According to the above descriptions, the computation of a backward slice or forward slice over the AIFG can be solved by using an usual depth-first or breadth-first graph traversal algorithm to traverse the graph by taking some port or role vertices of interest as the start point of interest.

Figure 5 shows a backward slice over the AIFG with respect to the slicing criterion $(\text{cashier}, V_c)$ such that $V_c = \{pv5, pv6, pv7\}$.

6.2 Computing an Architectural Slice

The slice S_g computed above is only a slice over the AIFG of an architectural specification, which is a set of vertices of the AIFG. Therefore we should map each element in S_g to the source code of the specification. Let $P = (C_m, C_n, c_g)$ be an architectural specification and $G = (V_{com}, V_{con}, Com, Con, Int)$ be the AIFG of P . By using the concepts of a reduced component, connector, and configuration introduced in Section 4, a slice $S_p = (C'_m, C'_n, c'_g)$ of an architectural specification P can be constructed in the following steps:

1. Constructing a reduced component c'_m from a component c_m by removing all ports such that their corresponding port vertices in G have not been included in S_g and unnecessary elements in the computation from c_m . The reduced components C'_m in S_p have the same relative order as the components C_m in P .
2. Constructing a reduced connector c'_n from a connector c_n by removing all roles such that their corresponding role vertices in G have not been included in S_g and unnecessary elements in the glue from c_n . The reduced connectors C'_n in S_p have the same relative order as their corresponding connectors in P .
3. Constructing the reduced configuration c'_g from the configuration c_g by the following steps:

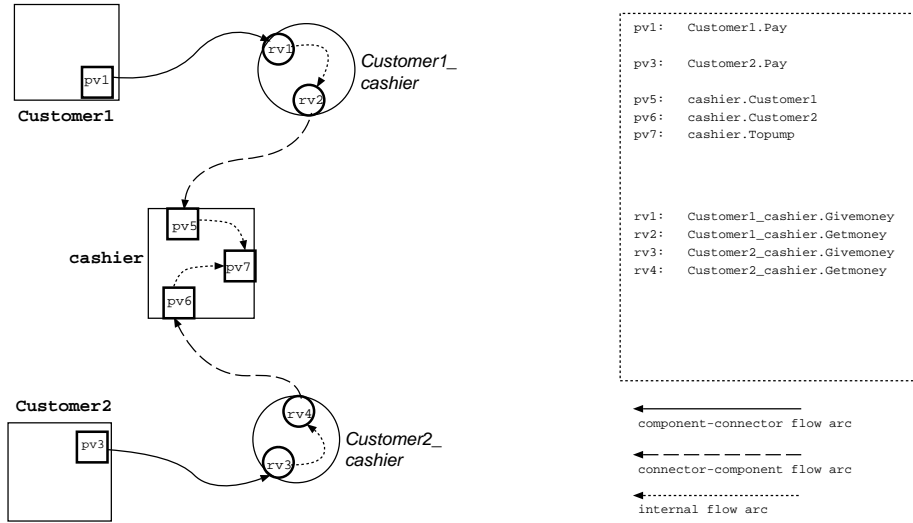


Figure 5: A slice over the AIFG of the architectural specification in Figure 2.

- Removing all component and connector instances from c_g that are not included in C'_m and C'_n .
- Removing all attachments from c_g such that there exists no two vertices v_1 and v_2 where $v_1, v_2 \in S_g$ and v_1 as v_2 represents an attachment.
- The instances and attachments in the reduced configuration in S_p have the same relative order as their corresponding instances and attachments in P .

Figure 6 shows a backward slice of the WRIGHT specification in Figure 2 with respect to the slicing criterion $(\text{cashier}, E)$ such that $E = \{\text{Customer1}, \text{Customer2}, \text{Topump}\}$ is a set of ports of component *cashier*. The small rectangles represent the parts of specification that have been removed, i.e., sliced away from the original specification. The slice is obtained from a slice over the AIFG in Figure 5 according to the mapping process described above. Figure 7 shows the architectural representation of the slice in Figure 6.

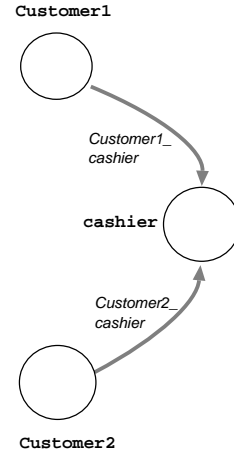


Figure 7: The architectural representation of the slice in Figure 6.

7 Related Work

7.1 Software Architecture Dependence Analysis

Perhaps, the most similar work with ours is that presented by Stafford, Richardson and Wolf [19], who introduced a software architecture dependence analysis technique, called *chaining* to support software architecture development such as debugging and testing. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related,

producing a chain of dependences similar to a slice in traditional slicing that can be followed during analysis. Although their consideration is similar to ours, there are still some differences between their work and ours. First, the slicing criteria are different. While Stafford, Richardson, and Wolf define a slicing criterion of an architectural specification as a set of ports of a component, we defined a slicing criterion as either a set of ports of a component or a set of roles of a connector of an architectural specification. This is because that in addition to modifying a component, in some cases, a maintainer may also want to modify a connector. Second, the types of architectural slices are different. Stafford, Richardson, and Wolf compute an architectural slice that in-

slice for programs written in imperative programming languages: (1) that variables and statements are concepts of the programming language in which program is written, and (2) that slices consist only of statements. For a language that does not have variables and statements, for example, a compiler specification language, traditional slicing does not make sense. To solve this problem, they introduced the generalized slicing as an extension of the traditional slicing by replacing variables with arbitrary named program entities and statements with arbitrary program constructs. This allows them to perform the slicing of non-imperative programs. Our work has a similar goal with theirs, but focuses specially on software architectures.

8 Concluding Remarks

We introduced a new form of slicing, named *architectural slicing* to aid architectural understanding and reuse. In contrast to the traditional slicing, architectural slicing is designed to operate on the architectural specification of a software system, rather than the source code of a program. Architectural slicing provides knowledge about the high-level structure of a software system, rather than the low-level implementation details of a program. In order to compute an architectural slice, we presented the *architecture information flow graph* to explicitly represent information flows in a formal architectural specification. Based on the graph, we gave a two-phase algorithm to compute an architectural slice.

While our initial exploration used WRIGHT as the architecture description language, the concept and approach of architectural slicing are language-independent. However, the implementation of an architectural slicing tool may differ from one architecture description language to another because each language has its own structure and syntax which must be handled carefully.

In architectural description languages, in addition to provide both a conceptual framework and a concrete syntax for characterizing software architectures, they also provide tools for parsing, displaying, compiling, analyzing, or simulating architectural specifications written in their associated language. However, existing language environments provide no tools to support architectural understanding, maintenance, testing, and reuse from an engineering viewpoint. We believe that some static analysis tools such as an architectural slicing tool introduced in this paper and an architectural dependence analysis tool [19, 26] should be provided by any ADL as an essential means to support these development activities.

As future work, we would like to extend our approach presented here to handle other constructs in WRIGHT language such as *styles* which were not considered here, and also to extend our approach to handle the slicing problem for other architecture description languages such as Rapide, ACME, and UniCon. To demonstrate the usefulness of our slicing approach, we are imple-

menting a slicer for WRIGHT architectural descriptions to support architectural level understanding and reuse. The next step for us is to perform some experiments to evaluate the usefulness of architectural slicing in practical development of software architectures.

Acknowledgements

The author would like to thank the anonymous referees for their valuable suggestions and comments on earlier drafts of the paper.

References

- [1] H. Agrawal, R. Demillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software-Practice and Experience*, Vol.23, No.6, pp.589-616, 1993.
- [2] R. Allen, "A Formal Approach to Software Architecture," PhD thesis, Department of Computer Science, Carnegie Mellon University, 1997.
- [3] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.
- [4] J. Beck and D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *Proceeding of the 15th International Conference on Software Engineering*, pp.509-518, Baltimore, Maryland, IEEE Computer Society Press, 1993.
- [5] J. Cheng, "Slicing Concurrent Programs – A Graph-Theoretical Approach," *Lecture Notes in Computer Science*, Vol.749, pp.223-240, Springer-Verlag, 1993.
- [6] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," *Proceedings of the Fourth Workshop on Program Comprehension*, Berlin, Germany, March 1996.
- [7] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.
- [8] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.
- [9] D. Garlan, R. Monroe, and D. Wile, "ACME: An Architecture Description Interchange Language," *Proceedings of CASCON'97*, November 1997.

- [10] C.A.R. Hoare, "Communicating Sequential Processes," Prentice Hall, 1985.
- [11] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, Vol.2, No.2, pp.47-57, 1985.
- [12] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.
- [13] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Veera, D. Bryan, and W. Mann, "Specification Analysis of System Architecture Using Rapide," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.336-355, April 1995.
- [14] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil, "Applying Static Analysis to Software Architectures," *Proceedings of the Sixth European Software Engineering Conference*, LNCS, Vol.1301, pp.77-93, Springer-Verlag, 1997. v
- [15] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Automated Support for Legacy Code Understanding," *Communications of ACM*, Vol.37, No.5, pp.50-57, May 1994.
- [16] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.
- [17] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transaction on Software Engineering*, Vol.21, No.4, pp.314-335, April 1995.
- [18] M. Shaw and D. Garlan, "Software Architecture: Perspective on an Emerging Discipline," Prentice Hall, 1996.
- [19] J. A. Stafford, D. J. Richardson, and A. L. Wolf, "Chaining: A Software Architecture Dependence Analysis Technique," Technical Report CU-CS-845-97, Department of Computer Science, University of Colorado, September 1997.
- [20] A. M. Sloane, J. Holdsworth, "Beyond Traditional Program Slicing," *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pp.180-186, January 1996.
- [21] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, Vol.3, No.3, pp.121-189, September, 1995.
- [22] F. Tip, J. D. Choi, J. Field, and G. Ramalingam, "Slicing Class Hierarchies in C++," *Proceedings of the 11th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.179-197, October, 1996.
- [23] M. Weiser, "Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method," PhD thesis, University of Michigan, Ann Arbor, 1979.
- [24] J. Zhao, J. Cheng and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," *Proc. of the COMPSAC'96*, pp.312-320, IEEE Computer Society Press, August 1996.
- [25] J. Zhao, J. Cheng, and K. Ushijima, "Slicing Concurrent Logic Programs," in T. Ida, A. Ohori and M. Takeichi (Eds.), *Second Fuji International Workshop on Functional and Logic Programming*, pp.143-162, World Scientific, 1997.
- [26] J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding," in M. Li (Ed.), *New Technologies on Computer Software*, pp.135-142, International Academic Publishers, September 1997.